AMENDMENTS

IN THE SPECIFICATION:

Please replace the title with the following amended title.

SYSTEM AND METHOD FOR PROCESSING BREAKPOINT EVENTS IN A CHILD PROCESS GENERATED BY A PARENT PROCESS ~~BEING MONITORED UNDER AN INSTRUMENTATION SCHEME~~

Please replace paragraph [0002] with the following amended paragraph.

[0037] While binary instrumentation leads to more precise results, the accuracy comes at some cost to the run-time performance of the instrumented program. Because the binary code of a program is modified, all interactions with the processor and the operating system can change significantly. For example, additional instructions and changes to a program's cache and paging behaviors can cause significant run-time performance degradation ~~increases from a few percent to 400%~~. Consequently, binary instrumentation is considered intrusive.

Please replace paragraph [0003] with the following amended paragraph.

[0038] Dynamic binary instrumentation allows program instructions to be changed on-the-fly and leads to a whole class of more precise run-time monitoring results. Unlike static binary instrumentation techniques that are applied over an entire program prior to execution of the program,[.] dynamic ~~Dynamic~~ binary instrumentation is performed at run-time of a program and only instruments those portions of an executable that are executed. Consequently, dynamic binary instrumentation techniques can significantly reduce the overhead imposed by the instrumentation process.

Please replace paragraph [0005] with the following amended paragraph.

[0005] A basic reason for the difficulty in testing the correctness of a program is that program behavior largely depends on the data on which the program operates and, in

the case of interactive programs, on the information (data and commands) received from a user. Therefore, even if exhaustive testing is impossible, as is often the case, program ~~testing and~~ verification is preferably conducted by causing the program to operate with some data. ~~In other words, by creating and performing what is defined as a "process" by software designers.~~

Please replace paragraph [0006] with the following amended paragraph.

[0006] Program verification encompasses execution of the program as a process "threads" to determine if the process develops in the correct way or if undesired or unexpected events occur. A "process" is commonly defined as an address space, one or more ~~a~~ control ~~thread~~ threads operating within the address space, and the set of system resources needed for operating with the ~~thread~~ threads. Therefore, a "process" is a logic entity consisting of the program itself, the data on which it ~~must operate~~ operates, the memory resources, and input / output resources. Executing a given program may lead to multiple processes being created. ~~Program verification and performance testing encompasses execution of the program as a process thread to test if the process develops in the correct way or if undesired or unexpected events occur.~~

Please replace paragraph [0010] with the following amended paragraph.

[0010] Symbolic debuggers have several limitations. First, they operate on only a single process at a time. ~~Second, the process to be tested must be activated by the parent process (the symbolic analysis process) and cannot be activated earlier. Consequently, the debugging of programs, which are activated at system start up, such as monitors, daemons, etc., is problematic. Furthermore,~~ Second, because the process to be tested is generated as a child of the symbolic analysis parent, and in a certain sense is the result of a combination of the symbolic analysis function/program with the program to be tested, the two processes must share or utilize the same resources. As a consequence, interactive programs that use masks and windows on a display device cannot be tested because they compete or interfere with the symbolic debugger in requiring access to the display device.

Please remove paragraph [0011].

Please replace paragraph [0012] with the following amended paragraph.

[0012] ~~Moreover, software development tools that use symbolic debuggers can encounter deadlock conditions that result from the standard execution of operating system level instructions.~~ One operating system that has gained widespread acceptance is the UNIX® operating system. UNIX® is a trademark of the American Telephone and Telegraph Company of New York, New York, U.S.A.

Please replace original paragraph [0015] with the following amended paragraph.

[0014] The popularity of the UNIX® operating system has led to the creation of numerous open source and proprietary variations such as LINUX®, HP-UX®, PRIMIX®, *etc.* LINUX® is a trademark of William R. Della-Croce, Jr. (individual) of Boston, Massachusetts, U.S.A. HP-UX®, is a trademark of the Hewlett-Packard Company, of Palo Alto, California, U.S.A. PRIMIX® is a trademark of Primix Solutions, Inc., of Watertown, Massachusetts, U.S.A. These and other variants of the UNIX® operating system inherently use the UNIX® operating system's logical I/O capabilities, pipes, and forks.

Please replace original paragraph [0016] with the following amended paragraph.

[0015] Software development tools can encounter a number of undesirable conditions when an instrumented process under test includes a "vfork" ~~"fork"~~ instruction. The operation of a "vfork" ~~"fork"~~ instruction in the UNIX® operating system involves spawning a new process, and then copying the process image of the parent (the process making the vfork ~~fork~~ call) to the child process (the newly spawned process). The parent process is suspended until the child process terminates. Consequently, the child process inherits any changes made to the address space of the parent process

before the child is created and any change made to the address space of the child process necessarily changes the address space of the parent process.

Please replace original paragraph [0017] with the following amended paragraph.

[0016] The dynamic instrumentation process changes the address space of a target application. More specifically, it inserts breakpoints into the function entry points in the text region. When an instrumented target process executes a vfork ~~fork~~ instruction, the child process inherits the text region containing the breakpoints from the parent (*i.e.*, the target) process.

Please replace original paragraph [0018] with the following amended paragraph.

[0017] FIG. 1 illustrates a deadlock condition. Deadlock condition 10 occurs between development tool 20, parent process 30, and child process 40 as follows. Development tool 20 instruments parent process 30 as indicated in function 22. ~~Development tool 20 assigns a~~ A process identifier (process ID) is assigned to the parent process 30 in function 24. Next, the development tool 20 monitors execution of the parent process using trace control as shown in function 26. Under the UNIX® operating system and its open source and proprietary variants, development tool 20 waits for trace events that include the process ID of the parent process as indicated in function 28. Development tool 20 cannot monitor child process 40, since child process 40 has not been created.

Please replace original paragraph [0019] with the following amended paragraph.

[0018] Once parent process 30 is created and started, parent process 30 runs nominally in accordance with its instructions until it encounters a ~~fork instruction (e.g.,~~ vfork instruction ~~fork or vfork)~~ as shown in function 32. Thereafter, as shown in function 34, ~~parent process 30 assigns~~ a process ID, different from ~~its own~~ the parent

process ID, is assigned to the child process, to identify the child process. In accordance with the vfork ~~fork~~ instruction, parent process 30 copies itself in its instrumented state to spawn child process 40 and generates a trace event which is received by development tool 20. Thereafter, as shown in function 38, parent process 30 is essentially suspended waiting for an indication that child process 40 has completed (*e.g.*, indicia of an exec or exit).

Please replace original paragraph [0020] with the following amended paragraph.

[0039] Once child process 40 is created by the vfork ~~fork~~ instruction in parent process 30, child process 40 runs nominally in accordance with its instructions until it encounters the vfork ~~fork~~ instruction shown in function 42. Thereafter, as shown in function 44, ~~child process 40 assigns~~ a new process ID, different from the child's process ID is assigned to the ~~its own process ID,~~ subsequent child process, to identify the subsequent child process. As illustrated in function 46, in accordance with the vfork ~~fork~~ instruction, child process 40 copies itself in its instrumented state to spawn the subsequent child process (not shown) and generates a trace event which is ignored by development tool 20 because development tool 20 is only looking for trace events from parent process 30.

Please replace original paragraph [0021] with the following amended paragraph.

[0040] Once the vfork ~~fork~~ instruction is encountered and processed in child process 40, the deadlock condition has occurred. Parent process 30 is suspended waiting for an indication that child process 40 has completed. Child process 40, which inherited trace control from parent process 30, waits for a process to handle the trace event generated at the time it executed the vfork ~~fork~~ instruction. Development tool 20 waits for a trace event from parent process 30.

Please replace original paragraph [0022] with the following amended paragraph.

[0041] Consequently it is desirable to have an improved apparatus, program, and method for handling vfork ~~fork~~ instruction induced deadlocks when using debugging techniques to instrument and monitor computer programs. It is also desirable to have an improved apparatus, program, and method for processing breakpoint events encountered during the execution of a child process. It is further desired to process breakpoint events encountered when executing a child process without further modifying the instrumented parent process.

Please replace original paragraph [0023] with the following amended paragraph.

[0042] An embodiment of a software tool includes logic configured to enable a child process that inherits the address space of a modified ~~instrumented~~ parent process to execute an unaltered version of the address space when the child process inherits the altered address space of the parent process.

Please replace original paragraph [0024] with the following amended paragraph.

[0043] An embodiment of a method for processing breakpoint events in a child process created from a parent process, when the parent process is modified ~~instrumented~~ by a software tool includes, storing unmodified ~~uninstrumented~~ parent process code replaced by each occurrence of a breakpoint inserted into the address space during modification ~~instrumentation~~ of the parent process, monitoring execution of a child process created by the parent process for an initial breakpoint in the address space, suspending execution of the child process in response to an initial breakpoint, replacing each occurrence of a breakpoint in the address space with the unmodified ~~uninstrumented~~ parent process code, and resuming execution of the child process.

Please replace original paragraph [0025] with the following amended paragraph.

[0044] Systems and methods for processing breakpoints in a child process generated by a modified ~~instrumented~~ parent process, as well as a method for run-time measuring a parent process modified ~~instrumented~~ by a software tool are illustrated by way of example and not limited by the implementations in the following drawings. The components in the drawings are not necessarily to scale, emphasis instead is placed upon clearly illustrating the principles used in controlling the execution of such a child process. Moreover, in the drawings, like reference numerals designate corresponding parts throughout the several views.

Please replace original paragraph [0027] with the following amended paragraph.

[0045] FIG. 2 is a functional block diagram of an embodiment of a computing device.

Please replace original paragraph [0031] with the following amended paragraph.

[0030] FIG. 6 is a flow chart illustrating an embodiment of a method for processing breakpoint events in a child process generated by an instrumented parent process, as well as a method for run-time measuring a parent process that includes a vfork ~~fork~~ instruction that can be implemented by the software monitor of FIG. 3.

Please replace original paragraph [0037] with the following amended paragraph.

[0036] If an event is requested by a process, the event mask of the thread is not examined. For the event mask of the thread to be significant, the process event must be unset. Similarly, if an event option is enabled in the process, the option for the thread is not considered. Event masks may be inherited across vfork ~~fork~~ instructions. For example, if tteo_proc_inherit is set, the child process inherits the event mask of

its parent. By default, threads stop when they receive a signal. If the signal being processed has its mask bit set, signal processing continues as though the process was not being traced. The traced thread is not stopped, and the tracing process is not notified of the signal. On the other hand, if the signal mask bit is not set for the signal being processed, the traced thread is stopped and the tracing process is notified via ttrace_wait.

Please replace original paragraph [0038] with the following amended paragraph.

[0037] As explained above, vfork ~~fork~~ instructions ~~(i.e., fork and vfork)~~ present a deadlock condition for debuggers that use known tracing facilities. For example, when the ttevt_fork event flag is set under the ttrace tracing facility both the parent thread and the initial thread in the child process stop (after the child process is marked as a traced process and adopts the parent thread's debugger). Both threads log the fact that they stopped in response to a ttevt_fork event. In the case of a vfork ~~fork~~ instruction where the ttev_vfork event flag is set, when the child process stops, its parent is asleep, and the child borrows the parent's address space until a call to exec or an exit (either by a call to exit or an abnormal termination of the child) takes place. Consequently, continuing the parent process before the child has completed results in an error.

Please replace original paragraph [0039] with the following amended paragraph.

[0038] In response, a modified debug interface includes a pre-fork event and associated processing adapted to multiple tracing facilities such as ttrace and/or ptrace. The modified debug interface and the associated methods described below enable a software tool to control the execution of a child process initiated by an instrumented parent process, where the parent process includes one or more vfork ~~fork~~ instructions. While the examples below are directed to an example where a parent process is instrumented, the present apparatus and methods are applicable to any modified parent process that includes a vfork ~~fork~~ instruction.

Please replace original paragraph [0040] with the following amended paragraph.

[0039] Turning now to the drawings, reference is made to FIG. 2, which illustrates a functional ~~function~~ block diagram of a computing device. Generally, in terms of hardware architecture, as shown in FIG. 2, computing device 200 includes a processor 210, memory 220, input/output device(s) 230, and network interface device(s) 240 that are communicatively coupled via local interface 250. The local interface 250 can be, for example but not limited to, one or more buses or other wired or wireless connections, as known in the art or that may be later developed. Local interface 250 may have additional elements, which are omitted for simplicity, such as controllers, buffers (caches), drivers, repeaters, and receivers, to enable communications. Further, local interface 250 may include address, control, and/or data connections to enable appropriate communications among the aforementioned components.

Please replace original paragraph [0044] with the following amended paragraph.

[0043] Software tool 300 and application(s) 224 include one or more source programs, executable programs (object code), scripts, or other collections each comprising a set of instructions to be performed. As will be explained in detail below, software tool 300 includes logic that controls the execution of application(s) 224. More specifically, software tool 300 includes logic that controls the execution of a child process or thread generated by an instrumented parent process found within application(s) 224 where the parent process or thread includes a vfork ~~fork~~ instruction. It should be well understood by one skilled in the art, after having become familiar with the teachings of the improved debug interface, that software tool 300 and application(s) 224 may be written in a number of programming languages now known or later developed that support the creation of child processes from a parent process using a vfork ~~fork~~ instruction. Moreover, software tool 300 and application(s) 224 may be stored across distributed memory elements in contrast with memory 220 shown in FIG. 2.

Please replace original paragraph [0049] with the following amended paragraph.

[0048] Process monitor 320 includes logic configured to identify and respond to events generated by specific processes. Process monitor 320 also includes logic that enables the software tool 300 to successfully execute a child process created by a vfork ~~fork~~ instruction in an instrumented parent process.

Please replace original paragraph [0053] with the following amended paragraph.

[0052] In addition, backpatch engine 360 includes logic configured to reinsert the breakpoint events inserted by the instrumentation engine 310 during the instrumentation process into the address space before the parent process is resumed after termination of the child process. Consequently, the address space is returned to its instrumented condition. With the breakpoint events present again in the address space the software tool 300 can measure, monitor, or otherwise control execution of the parent process after it executes the vfork ~~fork~~ instruction.

Please replace original paragraph [0060] with the following amended paragraph.

[0059] As further illustrated in the functional function diagram of FIG. 3, software tool 300 includes process monitor 320. Process monitor 320 includes logic for coordinating the collection of data during execution of parent process 370. When parent process 370 includes one or more vfork ~~fork~~ instructions, process monitor 320 ensures that data collected during execution of both the parent process 370 and its child process 372 are associated with the process responsible for generating the data. Alternatively, when it is desired to execute child process 372 without instrumentation, process monitor 320 ensures proper execution and data collection of parent process 370 and ensures proper execution of the vfork ~~fork~~ instruction created child process 372.

Please replace original paragraphs [0062] through [0065] with the following amended paragraphs.

[0061] Debug interface 330 includes logic for receiving and responding to the various trace system calls, events, and signals. In addition, debug interface 330 includes logic for generating instructions ~~334~~ 335. Instructions ~~335~~ 336 are in accordance with the underlying operating system 222. As illustrated in FIG. 3, debug interface 330 receives and responds to trace system calls, events, and signals 332 generated and sent by parent process 370, child process 372, and process monitor 320.

[0062] Operating system 222 as illustrated in FIG. 3, sends and receives instructions ~~334~~ 336 both to and from debug interface 330 via instruction interface 334. In addition, operating system 222 receives a vfork ~~fork~~ instruction 374 ~~(e.g., fork or vfork)~~ from parent process 370. As described in the UNIX® operating system and many of its proprietary and open source derivatives, vfork ~~fork~~ instruction 374 suspends execution of parent process 370 and generates child process 372 which contains a copy of the instrumented code and trace calls, events, and signals contained within parent process 370.

[0063] However, in addition to the other trace system calls, events, and signals 332, parent process 370 communicates a pre-fork event 375 to debug interface 330. Pre-fork event 375 includes indicia identifying child process 372 before it is created by a subsequently executed vfork ~~fork~~ instruction within parent process 370. The indicia includes at least a process identifier of the child process 372. Debug interface 330 further includes logic configured to recognize and respond to pre-fork event 375.

[0064] Software tool 300 uses process monitor 320 to trace the execution of parent process 370. When the child process 372 is generated and executes as a result of the vfork ~~fork~~ instruction in the parent process 370, process monitor 320 traces execution of child process 372. If child process 372 encounters a breakpoint event, a trace event is generated back to the process monitor 320 and child process 372 is suspended. Thereafter, backpatch engine 360 is configured to replace the initial breakpoint and any subsequent breakpoints in the address space with the appropriate original instruction bundle(s) 345 from process-image store 340 as illustrated by process arrow 342 and process arrow 362. As further indicated by process arrow 362, backpatch engine 360 is configured to generate list 355 for holding in breakpoint store 350.

List 355 contains each of the inserted breakpoints and an associated instruction address for each of the backpatched breakpoints. Once the address space reflects the uninstrumented state of the target process, process monitor 320 resumes execution of child process 372.

Please replace original paragraph [0068] with the following amended paragraph.

[0067] While the functional block ~~function~~ diagram presented in FIG. 3 illustrates software tool 300 as having a single centrally-located instrumentation engine 310 with co-located process monitor 320 and debug interface 330, it should be understood that the various functional elements of software tool 300 may be distributed across multiple locations in memory 220 and/or across multiple memory devices (not shown). It should be further understood that instrumentation engine 310 is not limited to dynamic binary instrumentation techniques and may include logic in accordance with binary instrumentation techniques (*i.e.*, logic that instruments all portions of the identified parent process 370) and statistical sampling.

Please replace original paragraphs [0069] through [0076] with the following amended paragraphs.

[0068] FIG. 4 is a flow chart illustrating an embodiment of a method for executing a parent process instrumented by a software tool to ensure execution of a child process when the parent process contains a vfork ~~fork~~ instruction. As illustrated in query 402, the parent process 370 begins by determining if a ~~fork or~~ vfork instruction is about to be executed by the parent process or thread. When it is determined that a ~~fork or~~ vfork instruction is about to be executed by the parent process or thread as indicated by the flow control arrow labeled "YES" that exits query 402, the parent process generates a pre-fork event as indicated in function 404. Next, as shown in function 406, the parent process sends the pre-fork event to the software tool responsible for instrumenting the parent.

[0069] Thereafter, as indicated in the wait loop formed by query 408 and wait function 410, execution of the parent process is suspended until after the parent receives

an indication from the software tool that the pre-fork event has been successfully processed. When the pre-fork event has been processed, as indicated by the flow control arrow labeled "YES" exiting query 408, the parent is activated and executes the vfork ~~fork~~ instruction as shown in function 412. Once the vfork ~~fork~~ instruction has been executed, the parent is suspended as indicated in function 414.

[0070] As indicated in the wait loop formed by query 416 and wait function 418, the parent process remains suspended until the parent receives an indication that the child process has terminated (*e.g.*, the child process generates an exec or an exit event). When the child process has terminated, as indicated by the flow control arrow labeled "YES" exiting query 416, the parent process resumes as shown in function 420. As indicated in query 422, the parent process continues until it terminates nominally and sends an exec event or fails and sends an exit event. As shown by the flow control arrow labeled "NO" exiting query 422, the parent is configured to report any future ~~fork or~~ vfork instructions by repeating the functions and queries described above.

[0071] Those skilled in the art will understand that the method for executing a parent process instrumented by a software tool to ensure execution of a child process when the parent process contains a vfork ~~fork~~ instruction illustrated in FIG. 4 is configured to generate and send a pre-fork event before executing a ~~fork or~~ vfork instruction. The parent process or thread may be implemented via multiple threads for controlling the performance of the desired functions. For example, a first thread may continuously identify when a vfork ~~fork~~ instruction is encountered in the instruction sequence. A second thread may intermittently be started to wait for an indication that the software tool has successfully processed the pre-fork event. A third thread may be responsible for handling trace events generated by the child process. These and other threads may be executed as may be desired by a parent process or thread to implement the various functions illustrated in FIG. 4.

[0072] Reference is now directed to the flow chart illustrated in FIG. 5, which illustrates an embodiment of a method for controllably switching a target process of a process monitor thread between an instrumented parent process and a child process generated by the parent process. In this regard, process monitor 320 begins with query 502 where it is determined if the child process has been successfully generated and started. If the result of query 502 indicates that the child process has not started successfully, process monitor 320 is configured to wait as indicated in function 504.

Next, query 506 is performed to determine if the parent process has received an indication that the vfork ~~fork~~ instruction failed. When the parent process receives an indication that the vfork ~~fork~~ instruction failed as indicated by the flow control arrow labeled "YES," the process monitor sets the active process identifier (PID) to the parent process' PID as shown in function 508. Otherwise, the process monitor returns to query 502. The determinations made in query 502 and query 504 are repeated to handle the case where an event documenting the creation of the child process as a result of the vfork ~~fork~~ instruction has not been received before the process monitor is started.

[0073] After the process monitor has set the PID to the parent process' PID, the process monitor continues by monitoring events and signals generated by the parent process as indicated by the monitoring loop formed by query ~~516~~ 514 and function ~~514~~ 516.

[0074] When the result of query 502 indicates that the child process has started successfully, process monitor 320 is configured to perform query 510 to determine if the parent process received an indication that the vfork ~~fork~~ instruction failed. In this way, the process monitor confirms that a child process was not generated by the parent process with the same PID identified in the pre-fork event. When the result of query 510 indicates that the vfork ~~fork~~ instruction failed, the process monitor is configured to notify the software tool 300 that the parent process has started two processes with the same PID as indicated in function 512. Otherwise, when query 510 indicates that the vfork ~~fork~~ instruction has not failed, the process monitor continues by executing the monitoring loop formed by query ~~516~~ 514 and function ~~514~~ 516. When query 502 indicates that the child process has started successfully and query 510 indicates that the parent process has not received an indication that the vfork ~~fork~~ instruction has failed, the target PID will reflect the PID of the child process generated by the vfork ~~fork~~ instruction executed by the parent process.

[0075] Reference is now directed to the flow chart illustrated in FIG. 6. In this regard, the various functions shown in the flow chart present both a method for processing breakpoint events in a child process created from a parent process, where the parent process is instrumented by a software tool, as well as a method for run-time measuring a parent process instrumented by a software tool, where the parent process includes a vfork ~~fork~~ instruction. Both methods may be realized by software tool 300. As

illustrated in FIG. 6, the method may begin by instrumenting a parent process as illustrated in function 602. Included in the process of instrumenting is storing an image of the uninstrumented set of instructions associated with each instrumented (*i.e.*, altered) function. The method for processing breakpoint events in a child process continues with query 604 where a determination is made if the child process has encountered a breakpoint. When the child process has not encountered an initial breakpoint as indicated by the flow control arrow labeled "NO" that exits query 604 the enters a wait loop formed by query 604 and wait function 606.

Please replace original paragraph [0079] with the following amended paragraph.

[0078] As illustrated in FIG. 6, the software tool 300 enters a monitoring loop formed by query 630, query 632, and wait function 634. When the parent process contains a subsequent vfork ~~fork~~ instruction as indicated by the flow control arrow labeled "YES" that exits query 630, the software tool 300 is programmed to repeat queries and functions 604 through 630 as may be required. Otherwise when no further vfork ~~fork~~ instructions are present in the address space of the parent process, as indicated by the flow control arrow labeled "NO" that exits query 630, the monitoring loop waits for the parent process to terminate.

Please replace original paragraph [0081] with the following amended paragraph.

[0080] Those skilled in the art will understand that while the methods illustrated in FIG. 6 are shown in a serial configuration a software tool 300 may include multiple threads for controlling the performance of the desired functions. For example, a first thread may continuously monitor and backpatch breakpoints encountered in vfork ~~fork~~-instruction created child processes. A parallel (*i.e.*, simultaneously executed) thread may continuously wait for an indication that a parent process is about to resume execution after termination of a vfork ~~fork~~-created child process. The parallel process may be configured to use a breakpoint list to reinstrument the address space of the

parent process prior to resuming execution of an instrumented parent process. These, and other, parallel threads may be executed as desired by software tool 300.

IN THE CLAIMS:

This listing of claims replaces all prior versions and listings of claims in the application.

1. (Currently Amended) A software tool, comprising:

logic configured to enable a child process that inherits the address space of a modified ~~instrumented~~ parent process to execute as if the child process was generated from an unaltered version of the address space when the child process inherits the altered address space of the parent process.

2. (Original) The software tool of claim 1, further comprising:

logic configured to enable execution of the altered address space after the child process terminates.

3. (Original) The software tool of claim 2, wherein the logic configured to enable execution of the altered address space further comprises:

a breakpoint store configured to receive a list of breakpoints inserted into the address space during modification of the parent process; and

a backpatch engine configured to receive the list, the backpatch engine further configured to reinsert the breakpoints in the address space.

4. (Original) The software tool of claim 3, wherein the backpatch engine is responsive to an event indicative of termination of the child process.

5. (Original) The software tool of claim 1, wherein the logic configured to enable a child process that inherits the address space of a parent process to execute further comprises:

a process image store configured to receive an original instruction bundle from the address space of the parent process when the original instruction bundle is altered with a breakpoint during binary instrumentation of the parent process; and

a backpatch engine configured to receive the original instruction bundle and replace the breakpoint with the original instruction bundle in the address space.

6. (Original) The software tool of claim 5, wherein the backpatch engine is responsive to an event indicative of a breakpoint encountered during execution of the child process.

7. (Currently Amended) A method for processing breakpoint events in a child process created from a parent process, wherein the parent process is modified by a software tool, the method comprising:

storing unmodified ~~uninstrumented~~ parent process code replaced by each occurrence of a breakpoint inserted into the address space during modification of the parent process;

monitoring execution of a child process created by the parent process for an initial breakpoint in the address space;

suspending execution of the child process in response to the initial breakpoint;

replacing each occurrence of a breakpoint in the address space with the unmodified ~~uninstrumented~~ parent process code; and

resuming execution of the child process.

8. (Original) The method of claim 7, wherein storing comprises retaining a copy of the instructions replaced by the breakpoint in a process image store.

9. (Original) The method of claim 7, wherein monitoring comprises executing a process monitor configured to respond to trace events generated by the child process.

10. (Currently Amended) A method for run-time measuring a parent process modified by a software tool, where the parent process includes a vfork ~~fork~~ instruction, the method comprising:

storing each occurrence of a breakpoint located in an address space associated with a parent process during modification of the parent process;

monitoring execution of the parent process for an indication that the parent process is about to resume execution after the termination of a child process generated in response to a vfork ~~fork~~ instruction, wherein the address space has been altered;

suspending execution of the parent process in response to the indication that the parent process is about to resume after termination of a child process generated in response to the vfork ~~fork~~ instruction;

restoring each breakpoint located in the address space during modification of the parent process to the address space; and

resuming execution of the parent process.

11. (Original) The method of claim 10, wherein storing comprises retaining a list including the breakpoint and an associated address in a breakpoint store.

12. (Original) The method of claim 11, wherein restoring further comprises applying the list.

13. (Original) The method of claim 12, wherein applying comprises inserting the breakpoint at the associated address within the address space.

14. (Original) The method of claim 10, wherein monitoring comprises executing a process monitor configured to respond to trace events generated by the parent process.

15. (Currently Amended) A computer-readable medium, comprising:

logic configured to enable a child process that inherits an address space of a modified ~~instrumented~~ parent process to execute as if the child process was generated from an unmodified address space; and

logic configured to enable execution of the modified ~~instrumented~~ parent process after the child process terminates.

16. (Currently Amended) The computer-readable medium of claim 15, wherein the logic configured to enable execution of the modified ~~instrumented~~ parent process further comprises:

logic configured to receive a list of breakpoints inserted into the address space during modification ~~instrumentation~~ of the parent process; and

logic configured to reinsert the breakpoints in the address space.

17. (Original) The computer-readable medium of claim 16, wherein the logic configured to reinsert the breakpoints is invoked in response to an indication that the child process has terminated.

18. (Original) The computer-readable medium of claim 15, wherein the logic configured enable a child process to execute further comprises:

logic configured to receive an original instruction bundle from the address space of the parent process when the address space is altered by inserting a breakpoint during binary instrumentation of the parent process; and

logic configured to replace the breakpoint with the original instruction bundle in the address space.

19. (Original) The computer-readable medium of claim 18, wherein the logic configured to replace the breakpoint with the original instruction bundle is responsive to an indication that child process encountered the breakpoint during execution.

IN THE ABSTRACT:

Please replace the following amended paragraph for the originally submitted Abstract.

[0086] A software tool includes logic configured to enable a child process that inherits the address space of a modified ~~an instrumented~~ parent process to execute an unaltered version of the address space when the child process inherits the altered address space of the parent process. A method for processing breakpoint events in a child process created from a parent process, when the parent process is modified ~~instrumented~~ by a software tool includes, storing unmodified ~~uninstrumented~~ parent process code replaced by each occurrence of a breakpoint inserted into the address space during modification ~~instrumentation~~ of the parent process, monitoring execution of a child process created by the parent process for an initial breakpoint in the address space, suspending execution of the child process in response to an initial breakpoint, replacing each occurrence of a breakpoint in the address space with the unmodified ~~uninstrumented~~ parent process code, and resuming execution of the child process.

IN THE DRAWINGS:

The attached drawings include changes to FIGs. 3-5, 7 and 8. The Appendix includes a replacement set of FIGs. 1-8, as well as annotated sheets showing the changes.

Amendments to FIG. 3:

"DEBUG INTERFACE 330" has been changed to comprise "INSTRUCTION GENERATOR 335," which further comprises "INSTRUCTION(S) 336."

Amendment to FIG. 4:

The label in query 402 has been changed to "VFORK ABOUT TO BE EXECUTED?"

Amendments to FIG. 5:

The label in queries 510 and 514 has been changed to "PARENT VFORK FAIL?"

Amendments to FIG. 7:

The label in block 702 has been changed to "STORE UNMODIFIED PARENT PROCESS CODE REPLACED BY EACH OCCURRENCE OF A BREAKPOINT INSERTED INTO THE ADDRESS SPACE DURING MODIFICATION."

The label in block 708 has been changed to "REPLACE EACH OCCURRENCE OF A BREAKPOINT IN THE ADDRESS SPACE WITH THE UNMODIFIED PARENT PROCESS CODE."

Amendments to FIG. 8:

The label in block 802 has been changed to "STORE EACH OCCURRENCE OF A BREAKPOINT LOCATED IN AN ADDRESS SPACE ASSOCIATED WITH A PARENT PROCESS DURING MODIFICATION OF THE PARENT PROCESS."

The labels in blocks 804 and 806 have been changed such that VFORK INSTRUCTION replaces FORK INSTRUCTION.

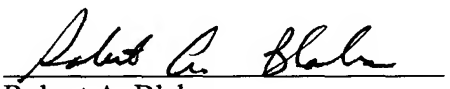
The label in block 808 has been changed such that MODIFICATION replaces INSTRUMENTATION.

REMARKS

Applicants request entry and consideration of the preceding amendments prior to examination of the application. Claims 1, 7, 10, 15, and 16 have been amended to clarify the subject matter that Applicants regard as the invention. Amendments to the specification and figures have been made to address informalities present in the originally submitted application. The amendments present no new matter to the present application.

Respectfully submitted,

THOMAS, KAYDEN, HORSTEMEYER & RISLEY, L.L.P.

By: 

Robert A. Blaha
Registration No. 43,502

APPENDIX

Substitute Specification (pages 1-21 and 26 of application).

Replacement FIGs. 1-8 (clean copy with changes).

Annotated FIGs. 1-8 (original drawings with redlines).

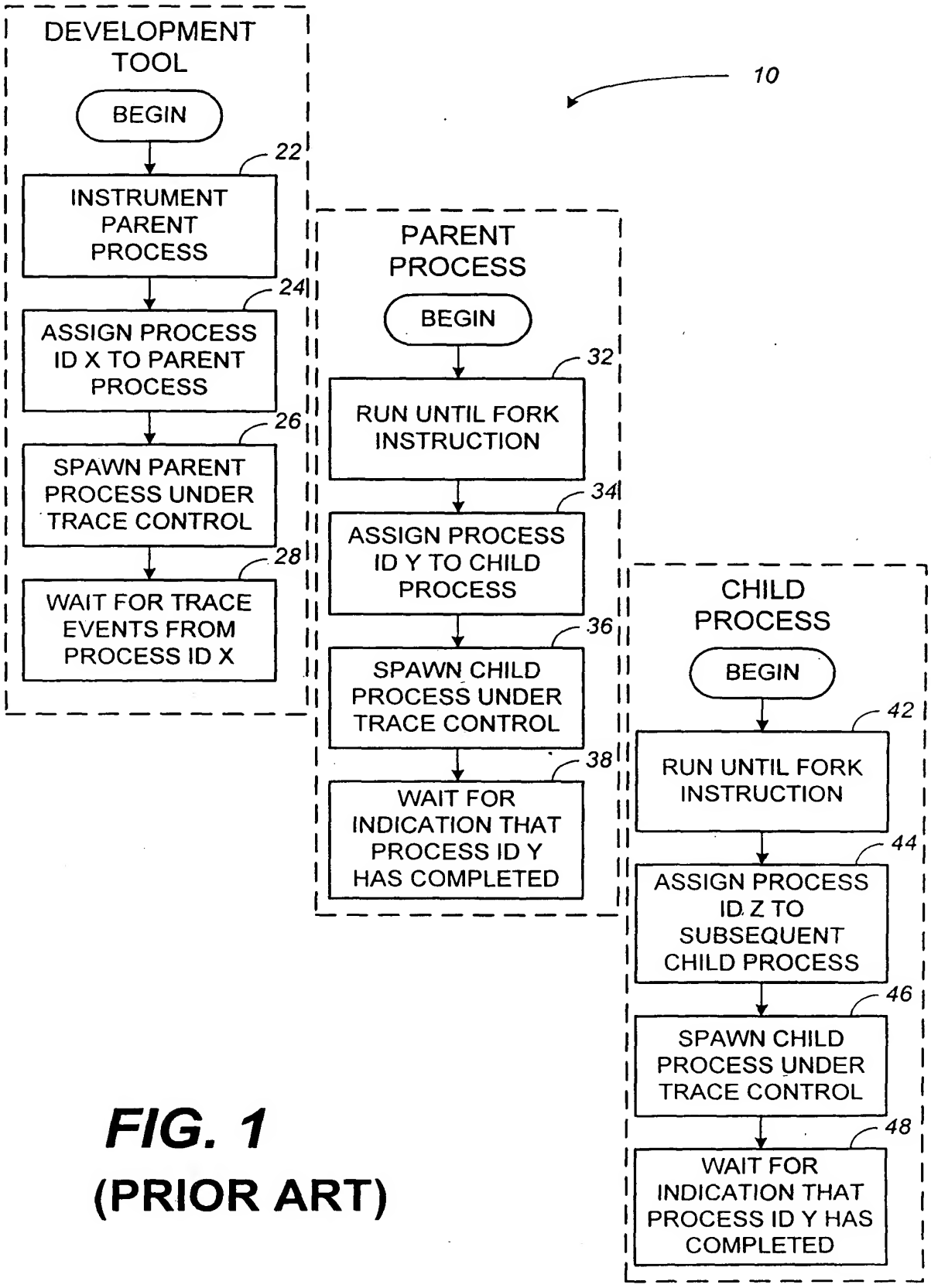


FIG. 1
(PRIOR ART)

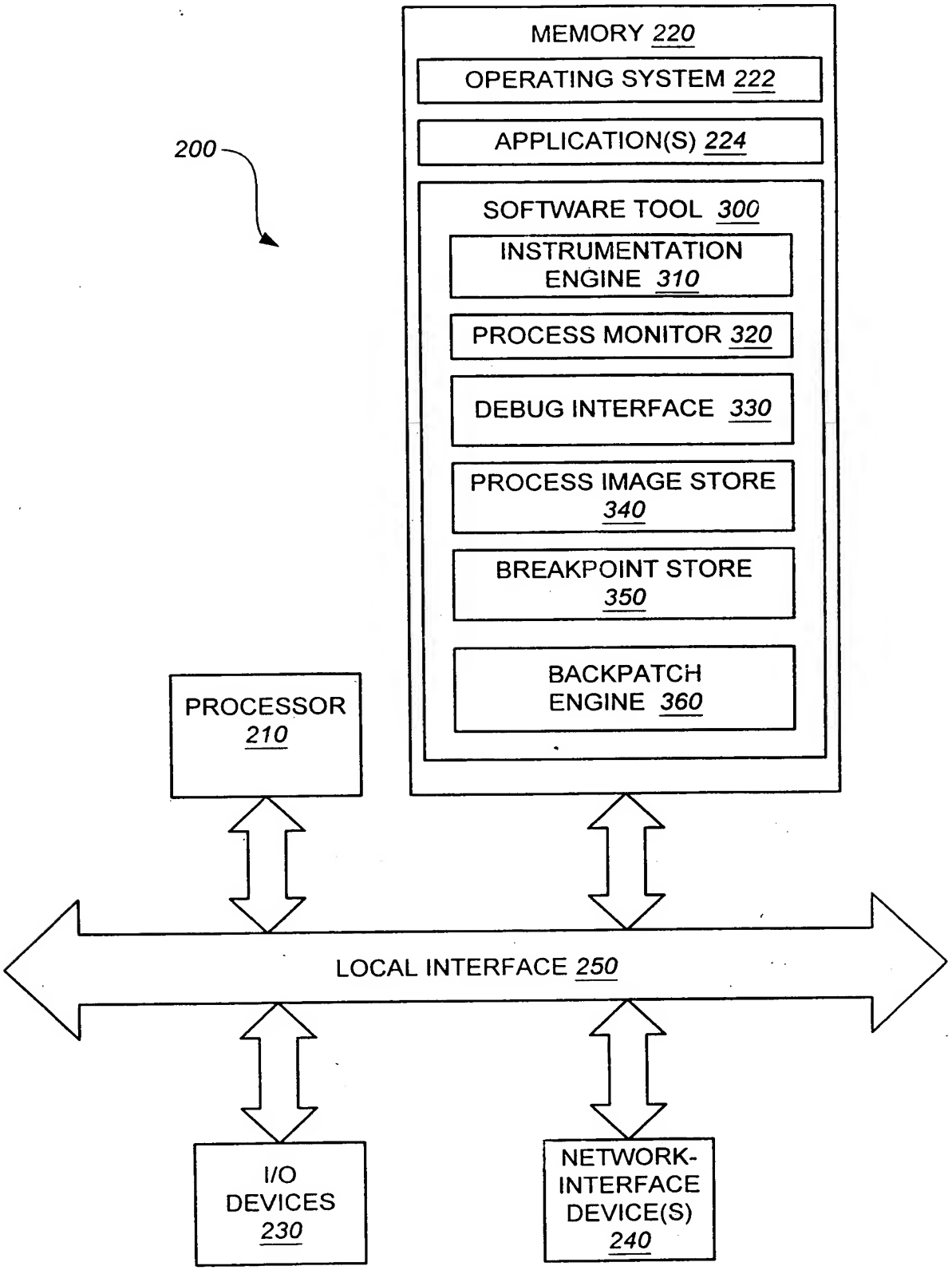


FIG. 2

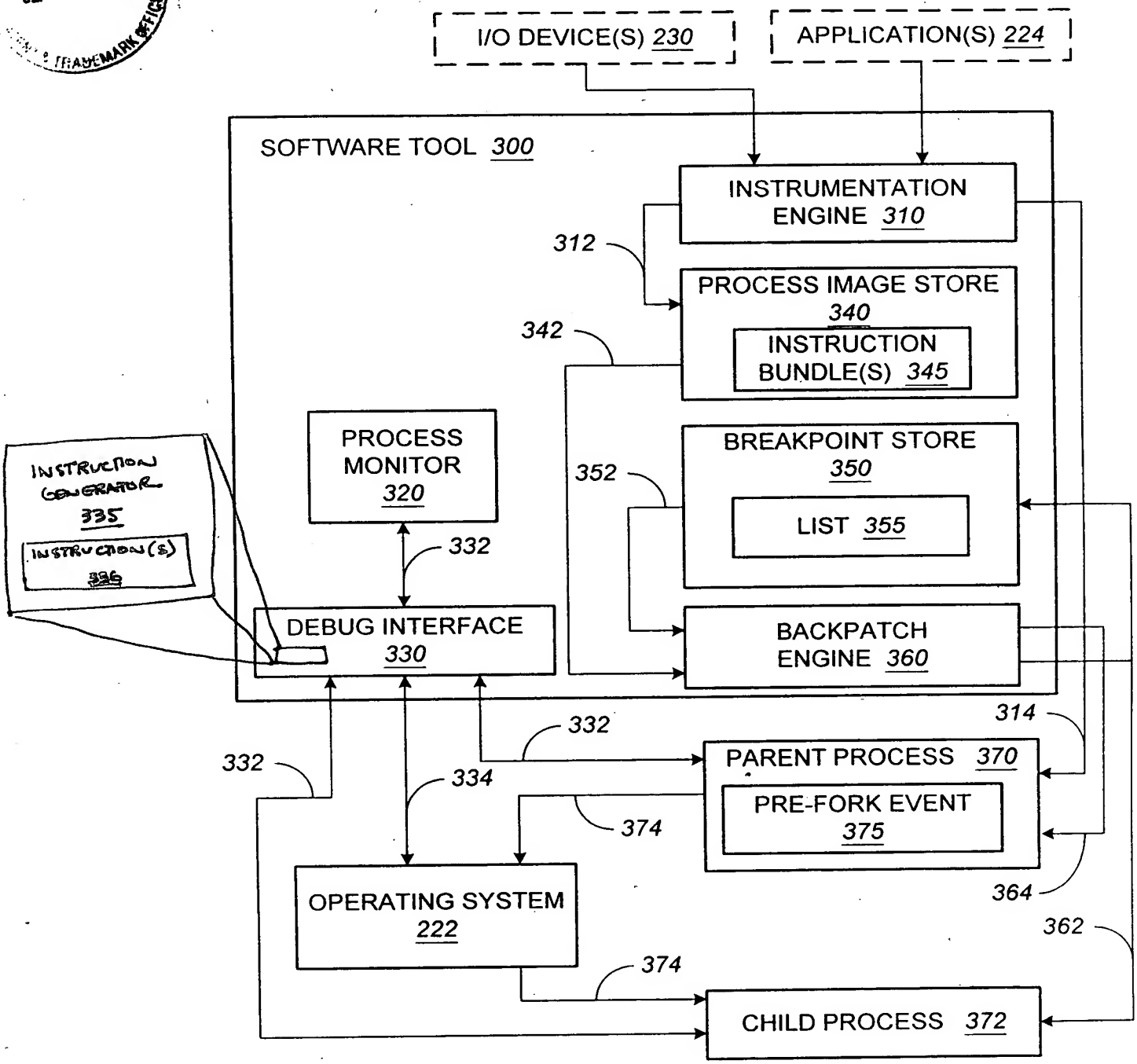


FIG. 3

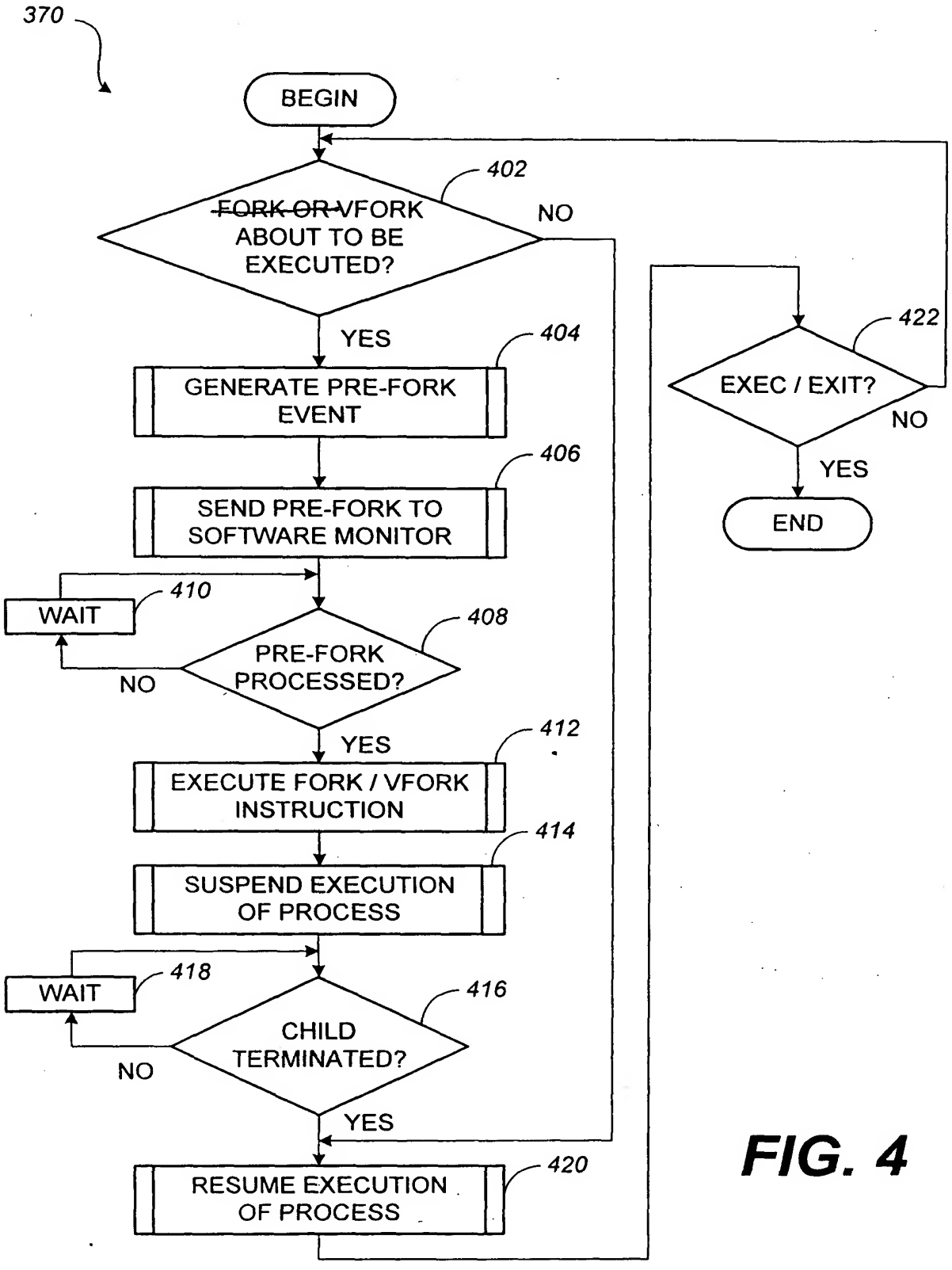


FIG. 4

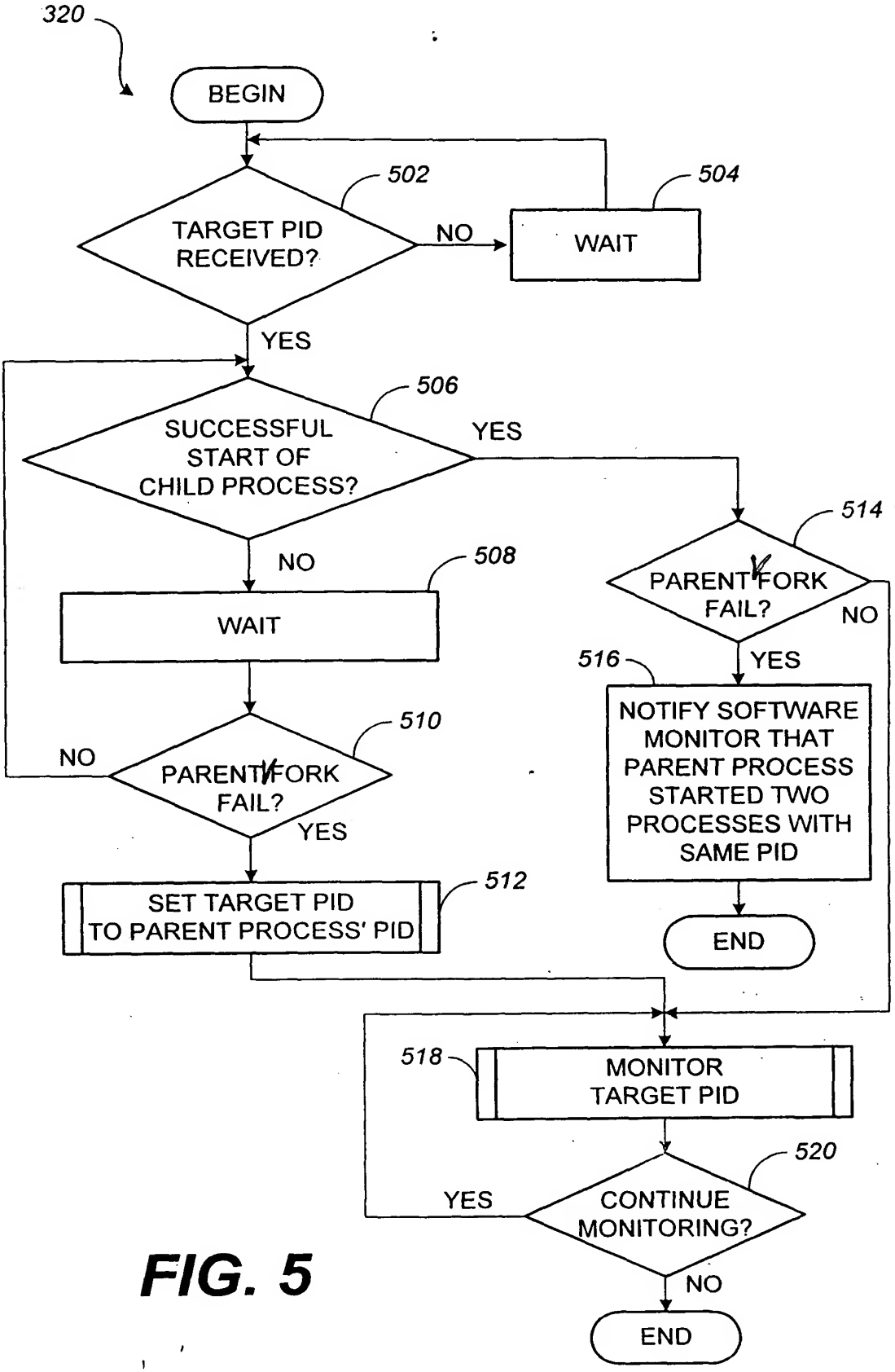


FIG. 5

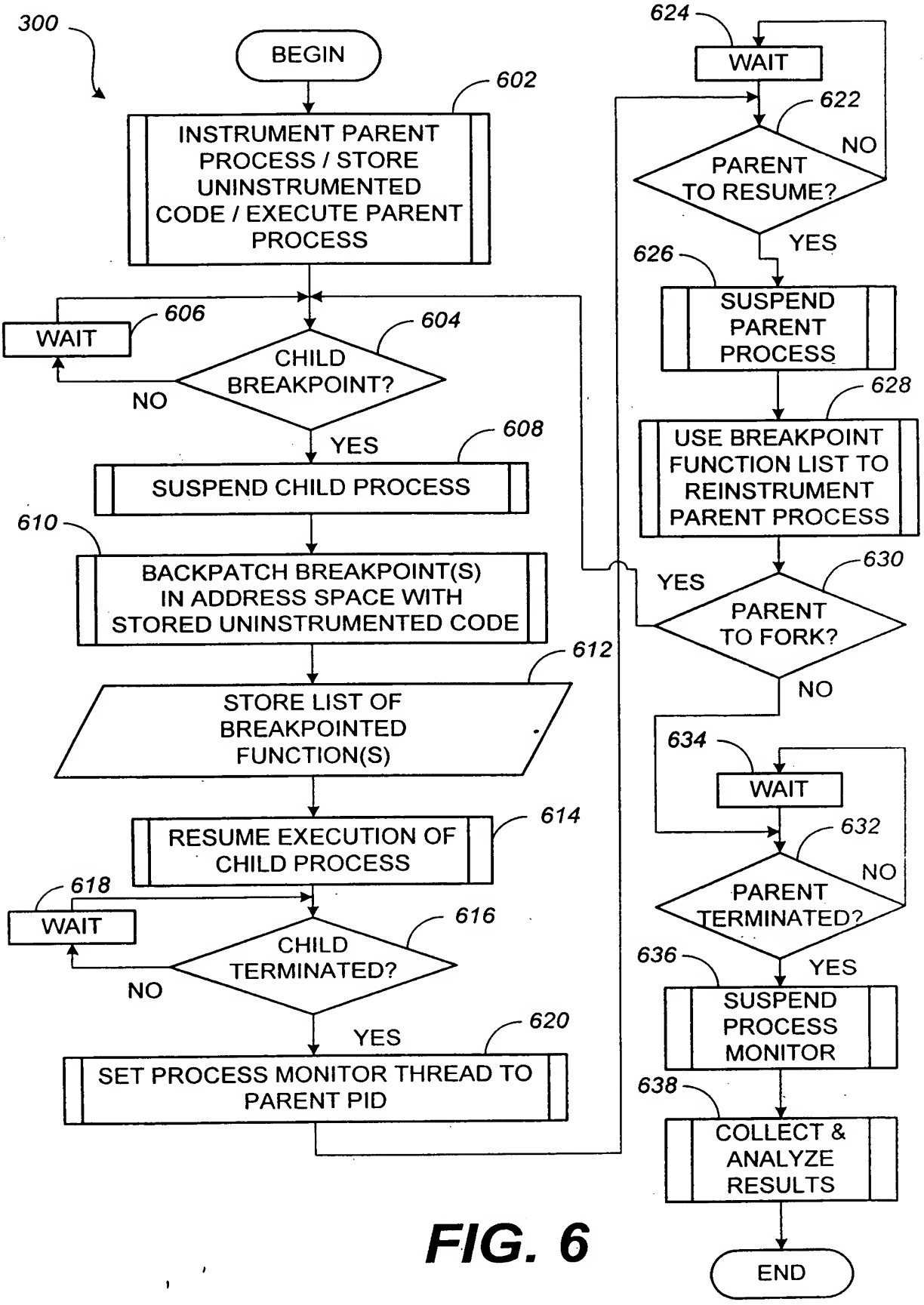


FIG. 6

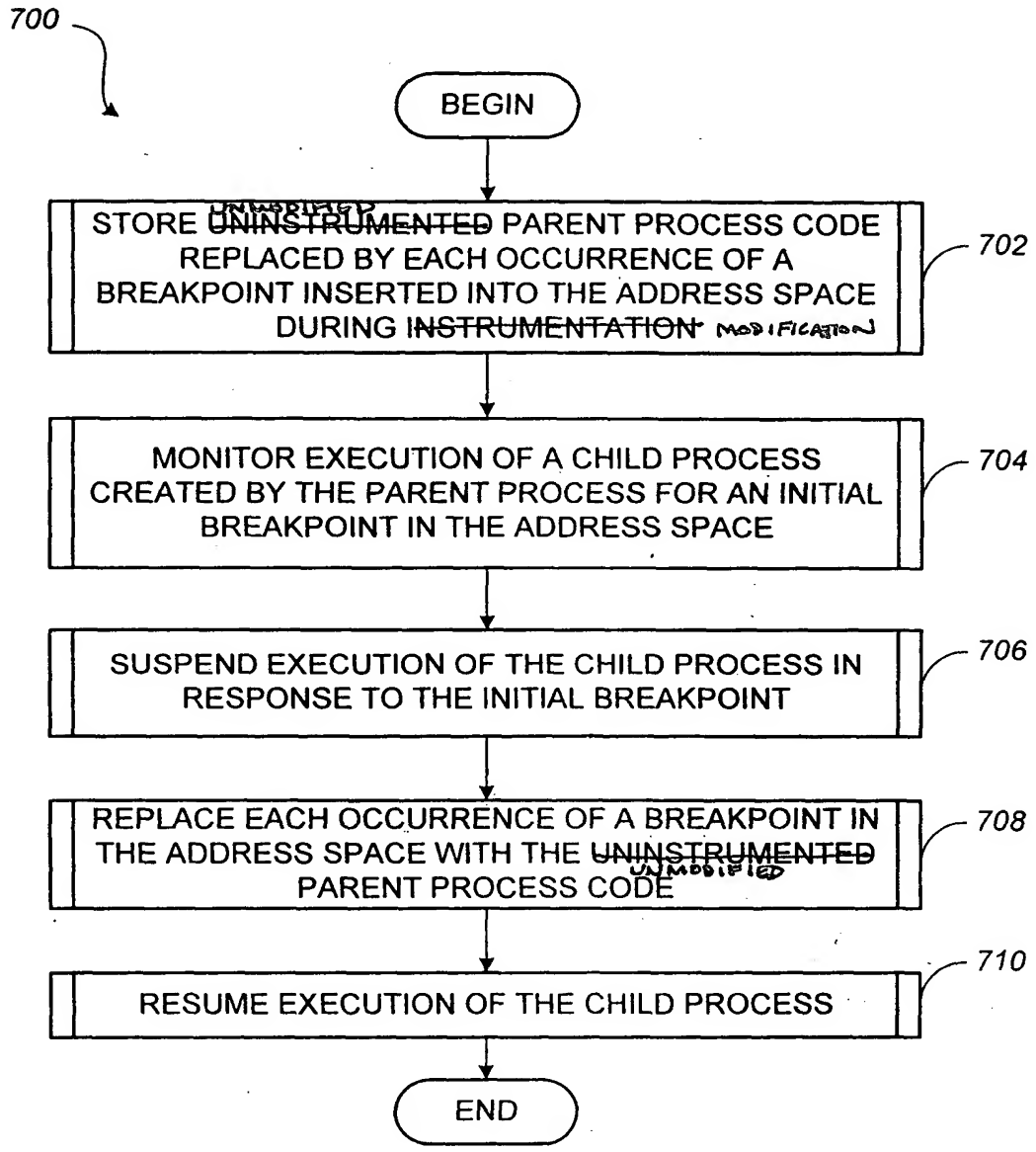


FIG. 7

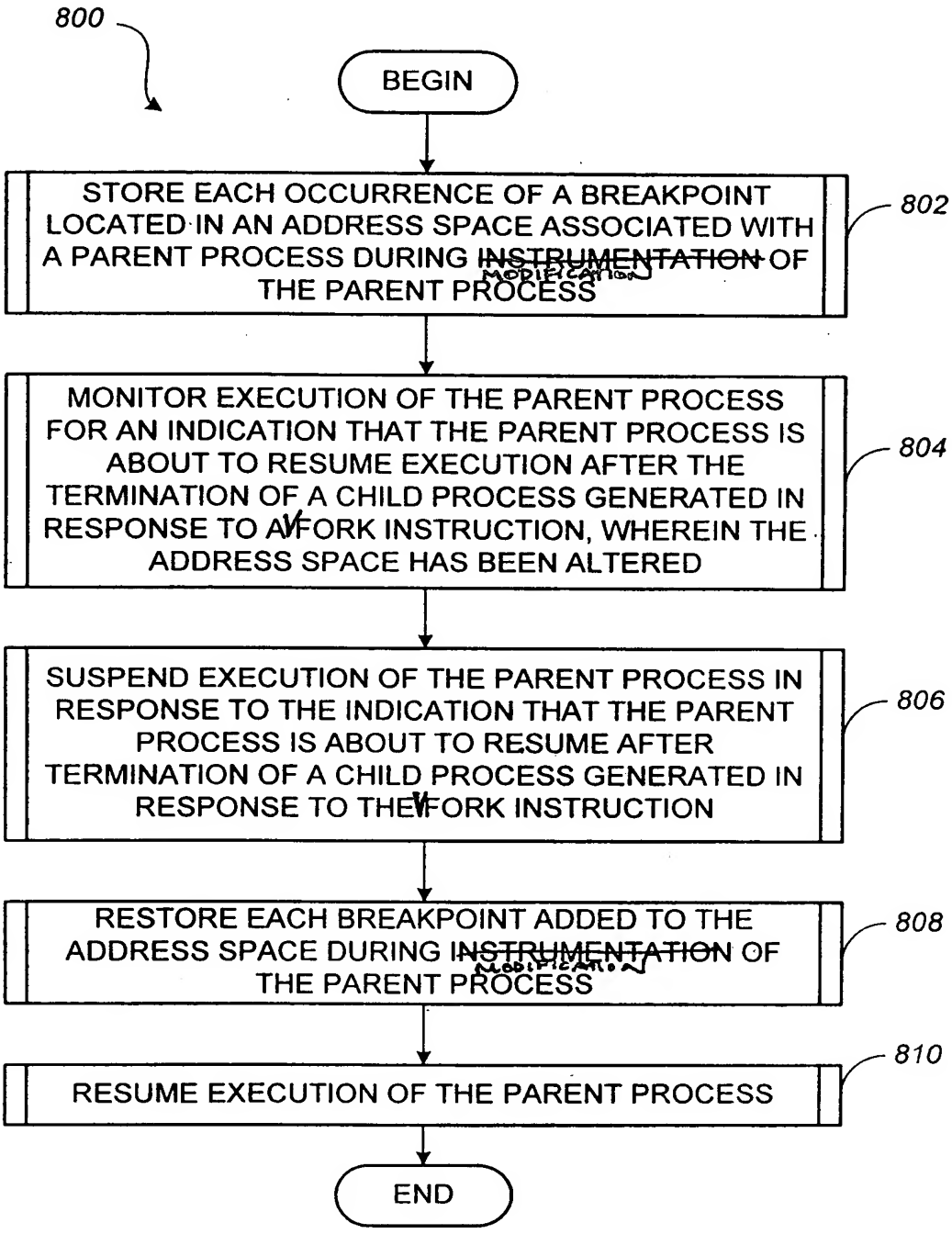


FIG. 8